MFC CDT Probability & Statistics: Part B - Assignment

Author: Niall Oswald

In [1]:
```python
# Import packages
import numpy as np
from matplotlib import pyplot as plt
from assignment.filters import *
from assignment.models import *

# Matplotlib setup
%matplotlib inline
plt.rcParams['figure.dpi'] = 150
```

The Model

Linear Gaussian State Space Models

A linear Gaussian state space model takes the form:

$$\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{q}_t, \qquad \mathbf{q}_t \sim N(0, \mathbf{Q}) \tag{1}$$
$$\mathbf{y}_t = \mathbf{H}\mathbf{x}_t + \mathbf{r}_t, \qquad \mathbf{r}_t \sim N(0, \mathbf{R}) \tag{2}$$

where $\mathbf{x}_t \in \mathbb{R}^m$ is the state at time t, and $y_t \in \mathbb{R}^n$ is the measurement. $\mathbf{A} \in \mathbb{R}^{m\times m}$ is the transition matrix with $\mathbf{q}_t \in \mathbb{R}^m$ the noise inherent to the model, which is assumed to be normally distibuted with zero mean and covariance matrix $\mathbf{Q} \in \mathbb{R}^{m\times m}$. Likewise $\mathbf{H} \in \mathbb{R}^{n\times m}$ is the measurement matrix and $\mathbf{r}_t \in \mathbb{R}^n$ is the noise in the error which is also assumed to be normally distributed with zero mean and covariance matrix $\mathbf{R} \in \mathbb{R}^{n\times n}$.

Noisy Resonator Model (Särkkä, 2013)

Consider a linear Gaussian state space model with transition and measurement matrices

$$\mathbf{A} = \begin{pmatrix} \cos\omega & \frac{\sin\omega}{\omega} \\ -\omega\sin\omega & \cos\omega \end{pmatrix}, \qquad \mathbf{H} = \begin{pmatrix} 1 & 0 \end{pmatrix}, \tag{3}$$

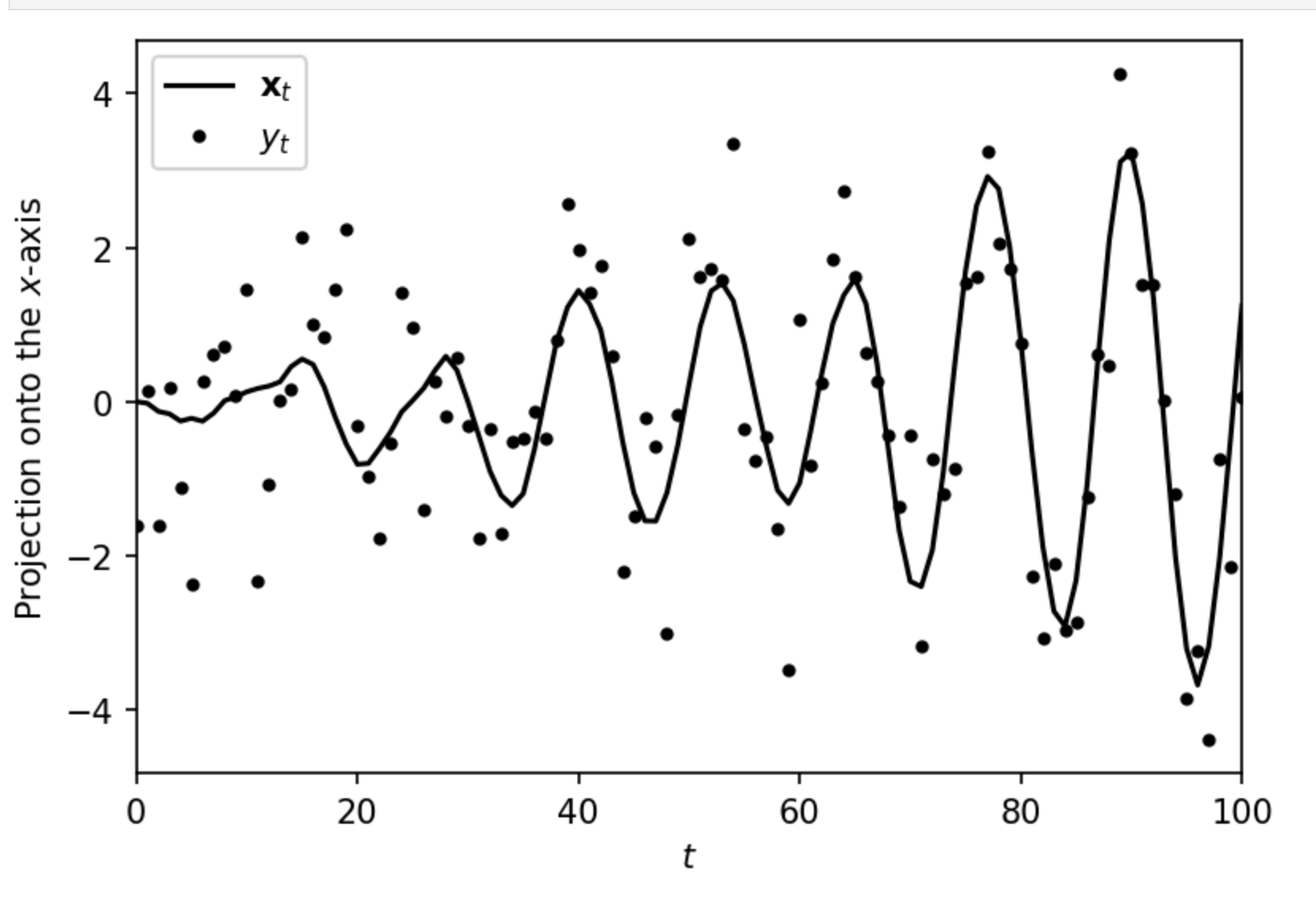
and covariance matrices

$$\mathbf{Q} = \begin{pmatrix} \frac{q^c(\omega - \cos\omega\sin\omega)}{2\omega^3} & \frac{q^c\sin^2\omega}{2\omega^2} \\ \frac{q^c\sin^2\omega}{2\omega^2} & \frac{q^c(\omega+\cos\omega\sin\omega)}{2\omega} \end{pmatrix}, \qquad \mathbf{R} = \begin{pmatrix} 1 \end{pmatrix}. \tag{4}$$

The model depends on two parameters, the angular velocity ω and the spectral density q^c. For the proceeding analysis we will generate data from this model, initialised at $\mathbf{x}_0 = (0, 0)$, using a fixed seed.

Data from this model is plotted below, taking the first entry of the state $\mathbf{x}_t$ and the observation y_t (which corresponds to a noisy projection of the same entry). Note that the covariance matrix $\mathbf{R}$ is taken to be far larger than that in Särkkä, 2013. This introduces far more noise into the observations, which provides a greater challenge for the filters.

In [2]:
```python
# Set the parameters
omega, q_c = 0.5, 0.01
n_iter = 100

# Run the model
model_iter = iter(resonator_model([omega, q_c]))

states, obs = np.zeros((n_iter + 1, 2)), np.zeros((n_iter + 1, 1))
for i in range(n_iter + 1):
    states[i], obs[i] = next(model_iter)

# Plot the data
plt.figure(figsize=(6, 4))
plt.plot(states[:, 0], "k-", label=r"$\mathbf{x}_t$")
plt.plot(obs[:, 0], "k.", label=r"$y_t$")
plt.xlim(0, n_iter)
plt.xlabel(r"$t$")
plt.ylabel(r"Projection onto the $x$-axis")
plt.legend()
plt.show()
```



Filtering

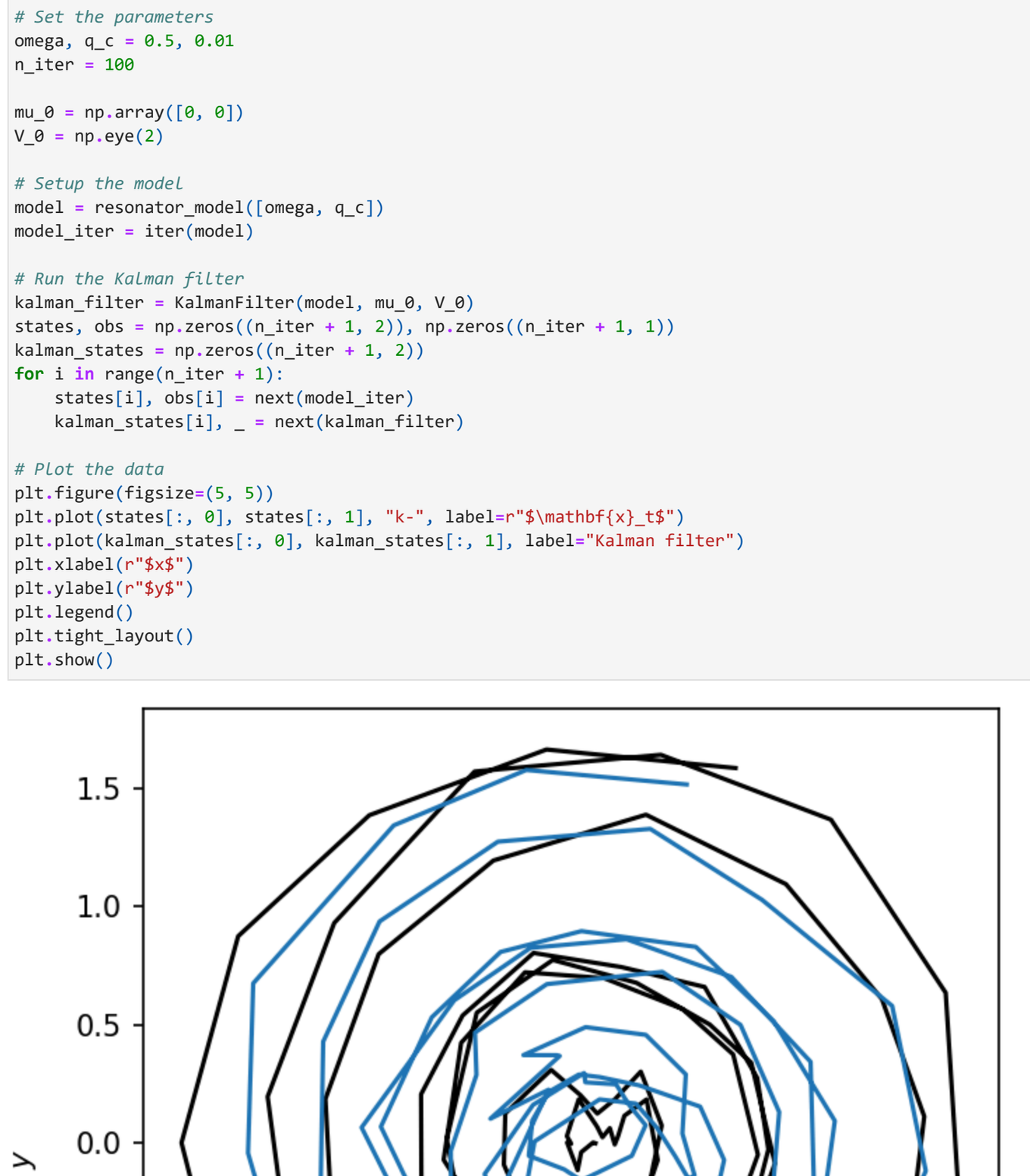
Now that we have introduced state space models, we will now turn to the question of filtering. Suppose now, that we only had access to the model parameters and the observation data y_t. Could we then construct an estimate for the true state $\mathbf{x}_t$ which lead to the observed data? There are two approaches that we will consider; each bringing their own benefits, namely Kalman filtering and particle filtering.

Kalman Filter

The Kalman filter is a purely deterministic approach. Given the same observation data and parameters, the filter will always provide the same estimates. The Kalman filter provides estimates for both the mean μ_t and the covariance $\mathbf{V}_t$ of the unobserved states $\mathbf{x}_t$. The filter is on-line, meaning that it will return estimates for the state at the current time based on observations as they are received. The simplest interpretation of the results is to take the mean μ_t as a point estimator for the state $\mathbf{x}_t$.

Below we plot the mean estimates μ_t alongside the unobserved states $\mathbf{x}_t$, where the x-axis denotes the first entry of $\mathbf{x}_t$ and the y-axis as the second entry. The results show that the filter is able to track the state with reasonable accuracy, despite only observing its first entry and with significant noise.

In [3]:
```python
# Set the parameters
omega, q_c = 0.5, 0.01
n_iter = 100

mu_0 = np.array([0, 0])
V_0 = np.eye(2)

# Setup the model
model = resonator_model([omega, q_c])
model_iter = iter(model)

# Run the Kalman filter
kalman_filter = KalmanFilter(model, mu_0, V_0)
states, obs = np.zeros((n_iter + 1, 2)), np.zeros((n_iter + 1, 1))
kalman_states = np.zeros((n_iter + 1, 2))
for i in range(n_iter + 1):
    states[i], obs[i] = next(model_iter)
    kalman_states[i], _ = next(kalman_filter)

# Plot the data
plt.figure(figsize=(5, 5))
plt.plot(states[:, 0], states[:, 1], "k-", label=r"$\mathbf{x}_t$")
plt.plot(kalman_states[:, 0], kalman_states[:, 1], label="Kalman filter")
plt.xlabel(r"$x$")
plt.ylabel(r"$y$")
plt.legend()
plt.tight_layout()
plt.show()
```



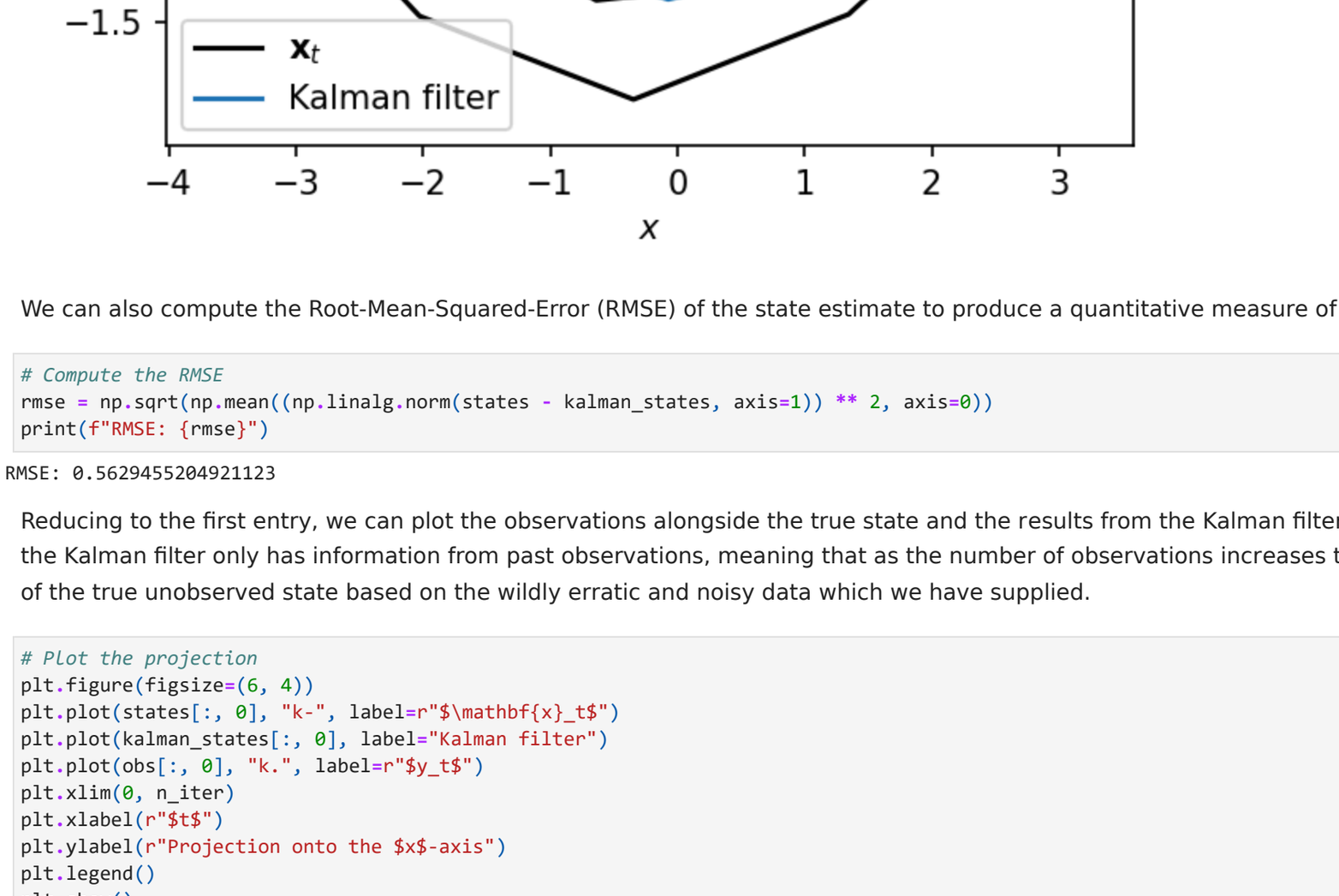
We can also compute the Root-Mean-Squared-Error (RMSE) of the state estimate to produce a quantitative measure of the accuracy of the filter.

In [4]:
```python
# Compute the RMSE
rmse = np.sqrt(np.mean((np.linalg.norm(states - kalman_states, axis=1)) ** 2, axis=0))
print(f"RMSE: {rmse}")
```

```
RMSE: 0.5629455204921123
```

Reducing to the first entry, we can plot the observations alongside the true state and the results from the Kalman filter. As time progresses we see that the mean state estimate increases in accuracy over time. This is what we expect, as the Kalman filter only has information from past observations, meaning that as the number of observations increases the estimate for the mean state improves. It is clear from the results that the Kalman filter is providing good estimates of the true unobserved state based on the wildly erratic and noisy data which we have supplied.

In [5]:
```python
# Plot the projection
plt.figure(figsize=(6, 4))
plt.plot(states[:, 0], "k-", label=r"$\mathbf{x}_t$")
plt.plot(kalman_states[:, 0], label="Kalman filter")
plt.plot(obs[:, 0], "k.", label=r"$y_t$")
plt.xlim(0, n_iter)
plt.xlabel(r"$t$")
plt.ylabel(r"Projection onto the $x$-axis")
plt.legend()
plt.show()
```

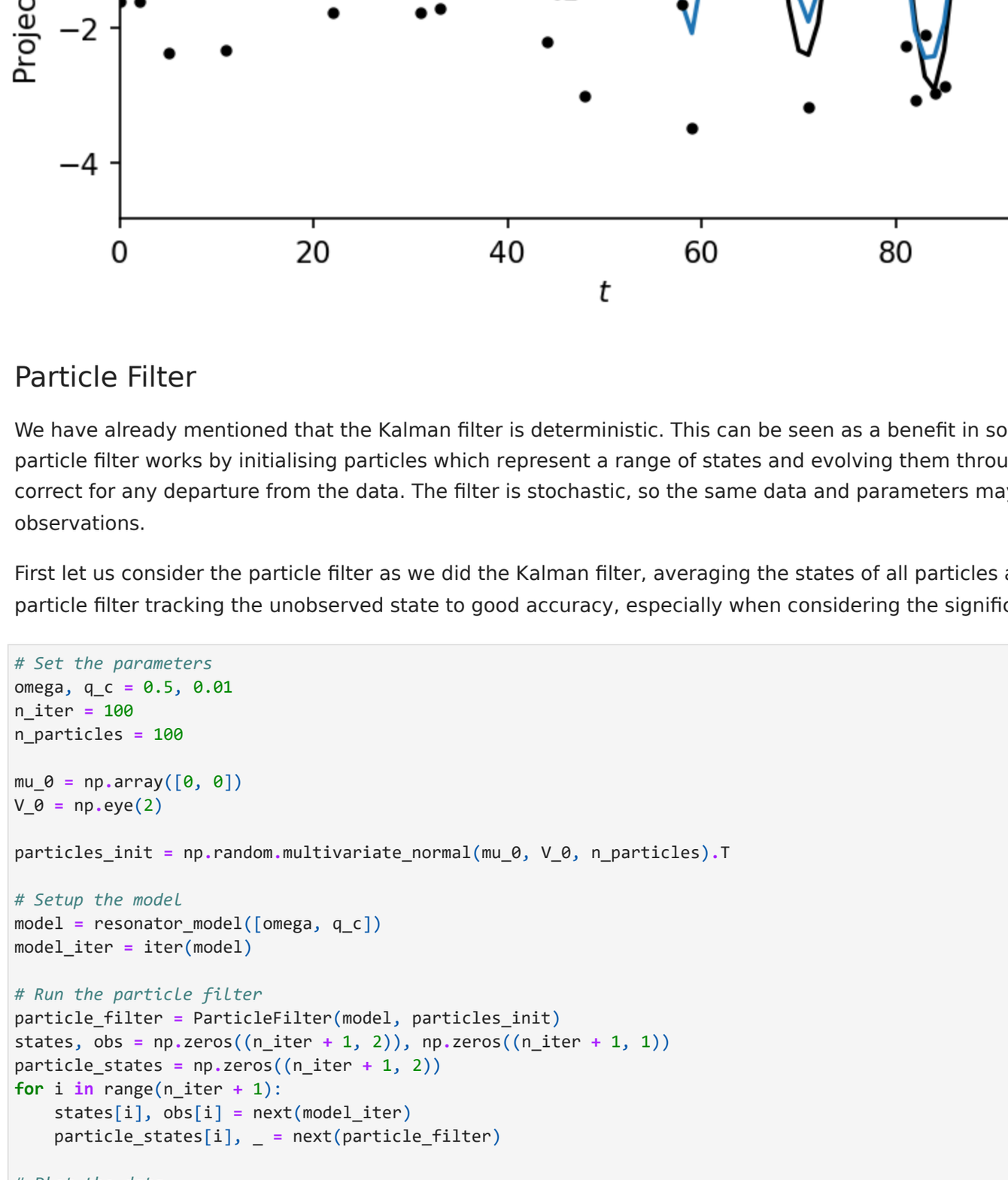


Particle Filter

We have already mentioned that the Kalman filter is deterministic. This can be seen as a benefit in some contexts, however it can also make it difficult to quantify the uncertainty of the filter. The particle filter is a departure from this. The particle filter works by initialising particles which represent a range of states and evolving them through the model. The likelihood of observations based upon these particles are then evaluated, leading to resampling of the particles to correct for any departure from the data. The filter is stochastic, so the same data and parameters may not necessarily lead to the same results. We will use this to our benefit later when attempting to fit model parameters to data observations.

First let us consider the particle filter as we did the Kalman filter, averaging the states of all particles at each time to produce a point estimate for the unobserved state. Again, we see very similar results to the Kalman filter, with the particle filter tracking the unobserved state to good accuracy, especially when considering the significant noise in the data.

In [6]:
```python
# Set the parameters
omega, q_c = 0.5, 0.01
n_iter = 100
n_particles = 100

mu_0 = np.array([0, 0])
V_0 = np.eye(2)

particles_init = np.random.multivariate_normal(mu_0, V_0, n_particles).T

# Setup the model
model = resonator_model([omega, q_c])
model_iter = iter(model)

# Run the particle filter
particle_filter = ParticleFilter(model, particles_init)
states, obs = np.zeros((n_iter + 1, 2)), np.zeros((n_iter + 1, 1))
particle_states = np.zeros((n_iter + 1, 2))
for i in range(n_iter + 1):
    states[i], obs[i] = next(model_iter)
    particle_states[i], _ = next(particle_filter)

# Plot the data
plt.figure(figsize=(5, 5))
plt.plot(states[:, 0], states[:, 1], "k-", label=r"$\mathbf{x}_t$")
plt.plot(particle_states[:, 0], particle_states[:, 1], "C1", label="Particle filter")
plt.xlabel(r"$x$")
plt.ylabel(r"$y$")
plt.legend()
plt.show()
```



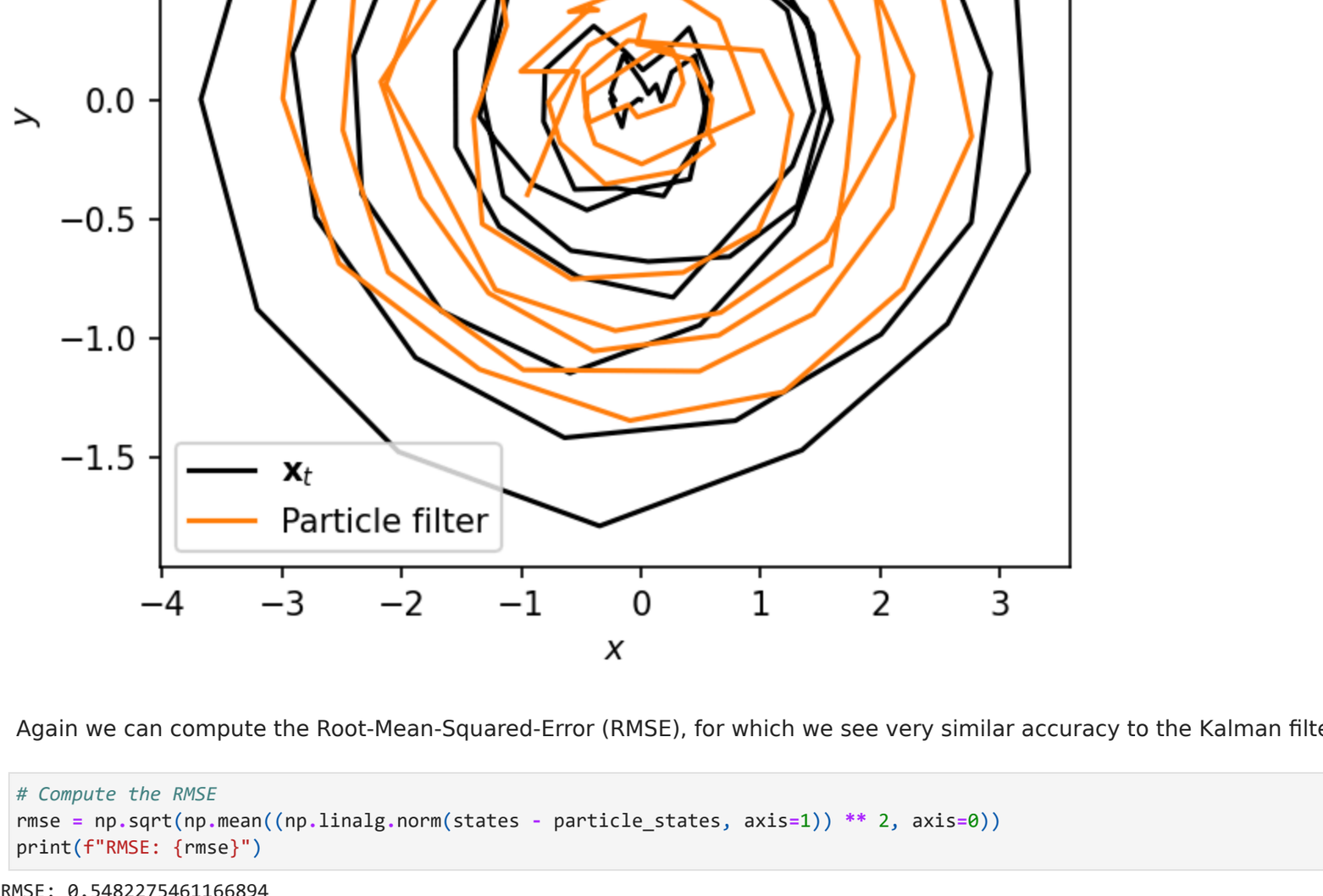
Again we can compute the Root-Mean-Squared-Error (RMSE), for which we see very similar accuracy to the Kalman filter, and in this specific instance is marginally lower.

In [7]:
```python
# Compute the RMSE
rmse = np.sqrt(np.mean((np.linalg.norm(states - particle_states, axis=1)) ** 2, axis=0))
print(f"RMSE: {rmse}")
```

```
RMSE: 0.5482275461166894
```

Projecting onto the first axis, we see a very similar story to the Kalman filter. Initially the data available to the filter is quite limited, leading to larger errors at the start of the window. As more data is observed these errors improve, eventually leading to a mean state estimate which matches the true state to good accuracy.

In [8]:
```python
# Plot the projection
plt.figure(figsize=(6, 4))
plt.plot(states[:, 0], "k-", label=r"$\mathbf{x}_t$")
plt.plot(particle_states[:, 0], color="C1", label="Particle filter")
plt.plot(obs[:, 0], "k.", label=r"$y_t$")
plt.xlim(0, n_iter)
plt.xlabel(r"$t$")
plt.ylabel(r"Projection onto the $x$-axis")
plt.legend()
plt.show()
```



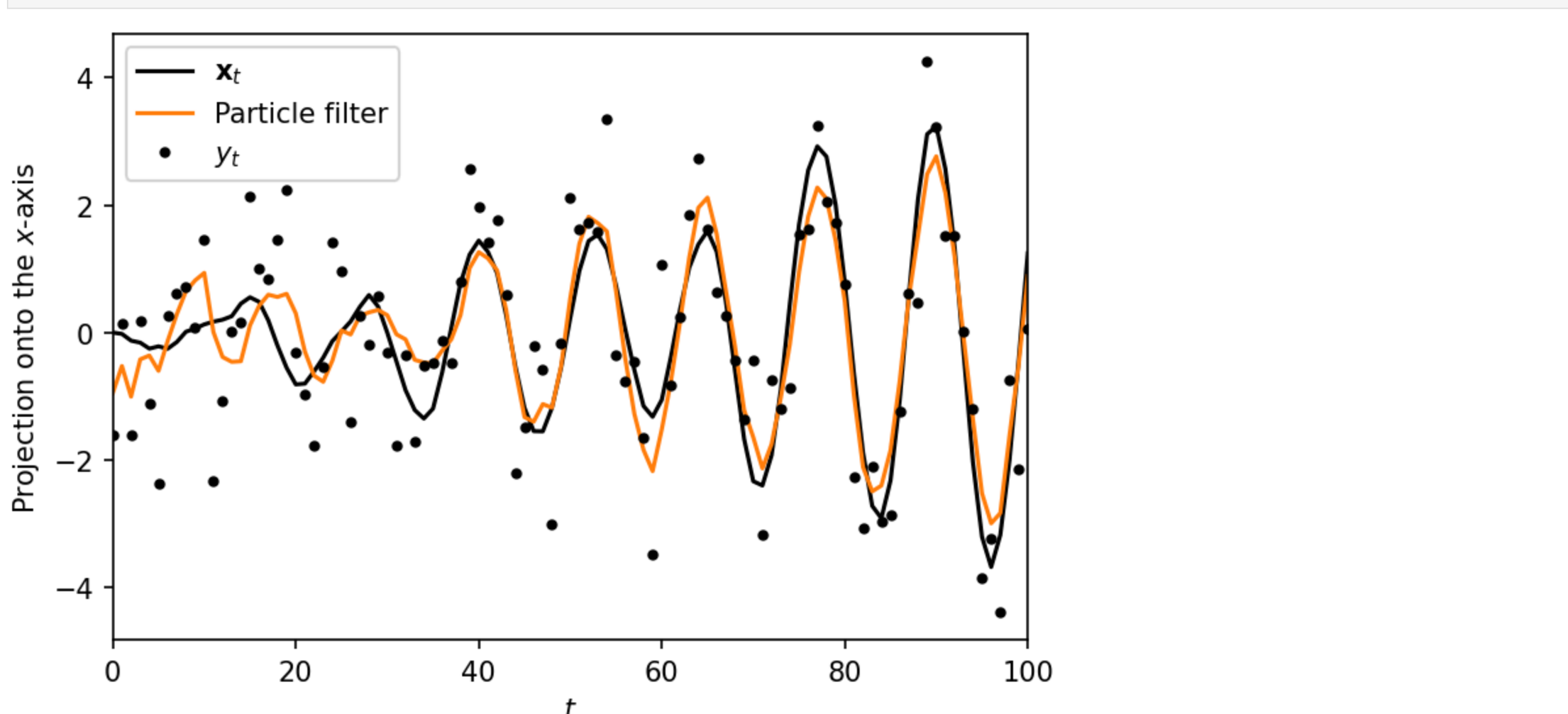
Marginal Likelihood Estimates

Suppose now that we wanted to fit a model to some data observations. Say we knew that the data was generated from a model which took the same form as before, however the parameters ω and q^c are unknown. Using the filters which we have presented, we can compute marginal likelihood estimates for the likelihood of the observations given a pair of parameters. We fix a "true model" with parameters $\omega = 0.5$ and $q^c = 0.01$ which are treated as unknown. We assume the form of the model given in (1, 2) and are provided with the observations y_t from the "true model". Given only this information, can we predict the parameters of the model used to generate this data?

In [9]:
```python
# Fix a model
true_model = resonator_model([omega, q_c])
```

Fixing one of the parameters to the true value, we can compute the marginal likelihood estimates for the other parameter based on both the Kalman and particle filters, plotted below. We see that there are clear maxima at the true parameter values. Note that the likelihoods obtained by the particle filter are noisy, this is not unexpected, as the particle filter is stochastic. As runs of the particle filter are repeated for each parameter value there will be some variance in the likelihood which we see as noise. This seems like a good start, but is it possible to find the maximum if we have no prior knowledge of either parameter?

In [10]:
```python
# Set the parameters
n_iter = 500
n_particles = 100

mu_0 = np.array([0, 0])
V_0 = np.eye(2)

particles_init = np.random.multivariate_normal(mu_0, V_0, n_particles).T

omega_values = np.linspace(0.4, 0.6, 200)
q_c_values = np.linspace(0.001, 0.02, 200)

# Compute the likelihood
omega_li = np.zeros((len(omega_values), 2))
for i, omega_val in enumerate(omega_values):
    for j, (filter, args) in enumerate(zip([KalmanFilter, ParticleFilter], [(mu_0, V_0), (particles_init,)])):
        omega_li[i, j] = filter.likelihood(true_model, resonator_model([omega_val, q_c]), n_iter, *args)

q_c_li = np.zeros((len(q_c_values), 2))
for i, q_c_val in enumerate(q_c_values):
    for j, (filter, args) in enumerate(zip([KalmanFilter, ParticleFilter], [(mu_0, V_0), (particles_init,)])):
        q_c_li[i, j] = filter.likelihood(true_model, resonator_model([omega, q_c_val]), n_iter, *args)

# Plot the likelihood
fig, axs = plt.subplots(1, 2, figsize=(8, 4))

axs[0].plot(omega_values, omega_li)
axs[0].set_xlabel(r"$\omega$")
axs[0].set_ylabel(r"log likelihood")

axs[1].plot(q_c_values, q_c_li)
axs[1].set_xlabel(r"$q^c$")
axs[1].set_ylabel(r"log likelihood")
axs[1].legend(["Kalman filter", "Particle filter"])

plt.tight_layout()
plt.show()
```



Model Fitting

Using the marginal likelihood estimates which we have plotted above, we can make use of an optimizer which attempts to fit a model by maximising the marginal likelihood. As we mentioned before, the particle filter is stochastic, so by repeating many runs of the particle filter we can not only average the results to achieve a more accurate estimate, but we can also produce uncertainty bounds for our prediction. Below we vary both the number of particles and the number of filter runs, producing uncertainty estimates for each parameter. By contrast, the Kalman filter is purely deterministic, producing a smooth marginal likelihood curve. This smooth curve is far easier to optimize using gradient ascent techniques compared to the noisy particle filter, however we also lose the uncertainty estimation afforded to us by the particle filter.

Another aside is that the Kalman filter is substantially faster to run, as the particle filter is required to update and resample many particles, compared to the Kalman filter which needs to only update the mean and covariance estimates. This difference in performance is exacerbated by the noise in the marginal likelihood of the particle filter, requiring either more particles or samples to converge to the optimum.

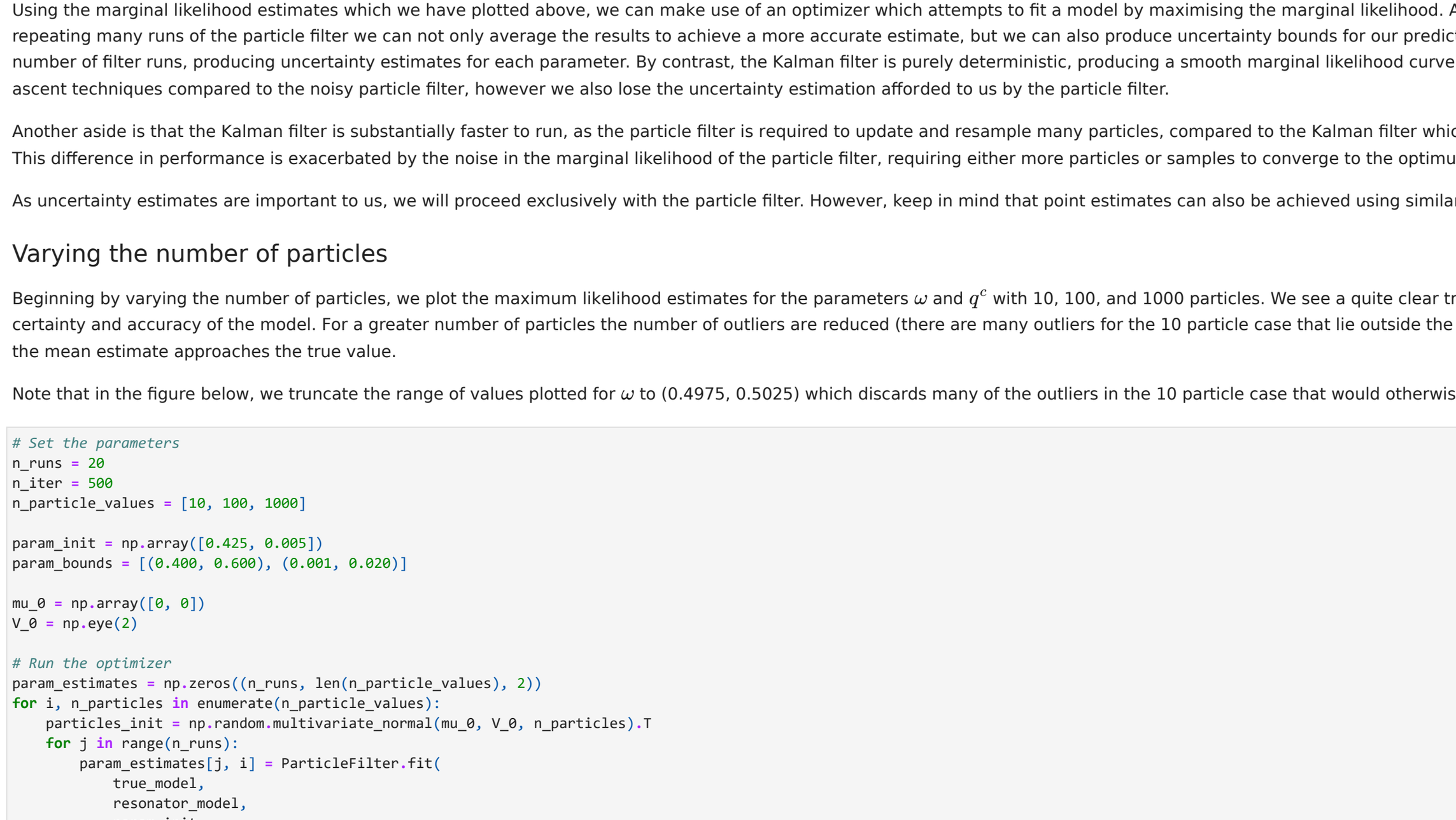
As uncertainty estimates are important to us, we will proceed exclusively with the particle filter. However, keep in mind that point estimates can also be achieved using similar methods (with far greater ease) using a Kalman filter as well.

Varying the number of particles

Beginning by varying the number of particles, we plot the maximum likelihood estimates for the parameters ω and q^c with 10, 100, and 1000 particles. We see a quite clear trend that increasing the number of particles increases the certainty and accuracy of the model. For a greater number of particles the number of outliers are reduced (there are many outliers for the 10 particle case that lie outside the limits of the figure), the spread of the estimates decrease, and the mean estimate approaches the true value.

Note that in the figure below, we truncate the range of values plotted for ω to (0.4975, 0.5025) which discards many of the outliers in the 10 particle case that would otherwise affect the clarity of the figure.

In [17]:
```python
# Set the parameters
n_runs = 20
n_iter = 500
n_particle_values = [10, 100, 1000]

param_init = np.array([0.425, 0.005])
param_bounds = [(0.400, 0.600), (0.001, 0.020)]

mu_0 = np.array([0, 0])
V_0 = np.eye(2)

# Run the optimizer
param_estimates = np.zeros((n_runs, len(n_particle_values), 2))
for i, n_particles in enumerate(n_particle_values):
    particles_init = np.random.multivariate_normal(mu_0, V_0, n_particles).T
    for j in range(n_runs):
        param_estimates[j, i] = ParticleFilter.fit(
            true_model,
            resonator_model,
            param_init,
            (particles_init,),
            n_iter,
            use_autograd=False,  # Autograd is not compatible with ParticleFilter
            bounds=param_bounds,
            method="Powell",  # Recommended for ParticleFilter to mitigate noise
        )

# Plot the estimates
fig, axs = plt.subplots(1, 2, figsize=(8, 4))

for i in range(2):
    axs[i].boxplot(param_estimates[..., i], tick_labels=n_particle_values)
    axs[i].set_xlabel("Number of particles")
    axs[i].set_ylabel("Parameter estimate")
    axs[i].set_title(f"MLE estimates for {[r"$\omega$", r"$q^c$"][i]}")


axs[0].set_ylim(0.4975, 0.5025)  # Cutoff some outliers for omega

plt.tight_layout()
plt.show()
```



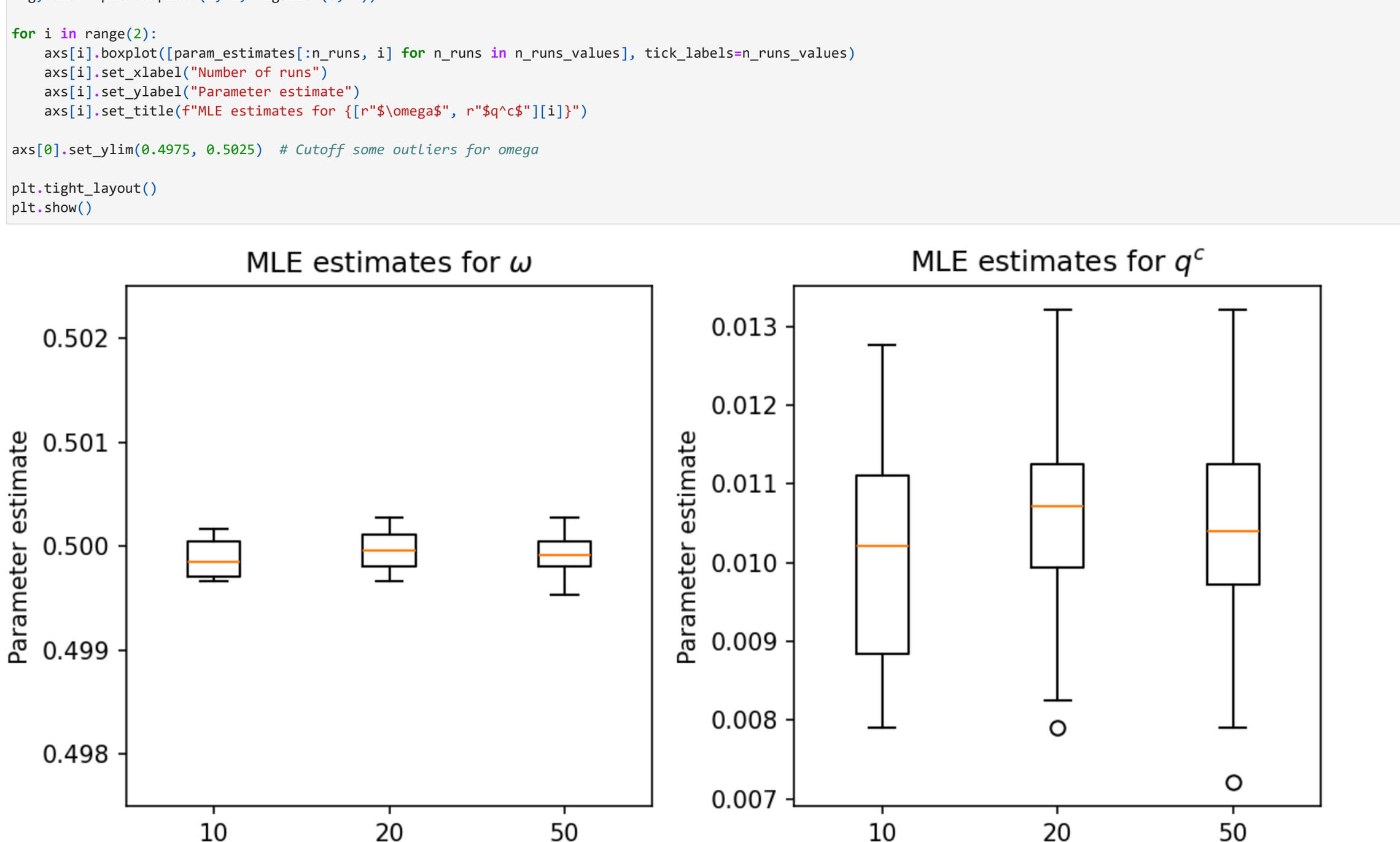
Varying the number of Monte Carlo runs

Keeping the number of particles fixed to 100, we now consider the effect of increasing the number of samples (this equates to full runs of the particle filter). Here we see a very different trend. Increasing the number of samples has little effect on the accuracy or precision of the parameter estimates, however it does somewhat correct the bias and skewness in the estimates. This is not unanticipated, as increasing the number of samples will simply give us a better estimate of the distribution of parameters estimated by a 100 particle filter. In comparison to before, where increasing the number of particles produced a more accurate and lower spread estimate of the parameter distribution from which we take only 20 samples.

Clearly by increasing the number of samples we will understand better the distribution of parameter estimates produced by a particle filter, however increasing the number of particles will produce more accurate and precise estimates from which we can sample the distribution. It is important to consider both when fitting a model, as each can be a source of error. For example there is little benefit to running a single sample of a 1,000,000 particle filter as similar results could be achieved by using a Kalman filter instead. Likewise running a single particle filter 1,000,000 times is unlikely to produce precise or accurate estimates.

Note that in the figure below, we again truncate the limits of ω to (0.4975, 0.5025) in order to remove many of the outliers for the 20 sample case.

In [18]:
```python
# Set the parameters
n_runs_values = [10, 20, 50]
n_iter = 500
n_particles = 100

param_init = np.array([0.425, 0.005])
param_bounds = [(0.400, 0.600), (0.001, 0.020)]

mu_0 = np.array([0, 0])
V_0 = np.eye(2)

particles_init = np.random.multivariate_normal(mu_0, V_0, n_particles).T

# Run the optimizer
param_estimates = np.zeros((n_runs_values[-1], 2))
for i in range(n_runs_values[-1]):
    param_estimates[i] = ParticleFilter.fit(
        true_model,
        resonator_model,
        param_init,
        (particles_init,),
        n_iter,
        use_autograd=False,  # Autograd is not compatible with ParticleFilter
        bounds=param_bounds,
        method="Powell",  # Recommended for ParticleFilter to mitigate noise
    )

# Plot the estimates
fig, axs = plt.subplots(1, 2, figsize=(8, 4))

for i in range(2):
    axs[i].boxplot([param_estimates[:n_runs, i] for n_runs in n_runs_values], tick_labels=n_runs_values)
    axs[i].set_xlabel("Number of runs")
    axs[i].set_ylabel("Parameter estimate")
    axs[i].set_title(f"MLE estimates for {[r"$\omega$", r"$q^c$"][i]}")

axs[0].set_ylim(0.4975, 0.5025)  # Cutoff some outliers for omega

plt.tight_layout()
plt.show()
```



Summary

In this report we have considered two methods of filtering applied to linear Gaussian state space models. We have explored the benefits of each approach, with Kalman filtering providing a faster algorithm to produce point estimates and leveraging the stochastic nature of the particle filter to produce uncertainty estimates when fitting models. Overall both techniques have been broadly successful for the model we have considered, accurately predicting unknown parameters based upon highly noised data.